An Embedded Domain Specific Language to Model, Transform and Quality Assure Business Processes in Business-Driven Development

Luana Micallef and Gordon J. Pace
Department of Computer Science, University of Malta

Abstract

In Business-Driven Development (BDD), business process models are produced by business analysts. To ensure that the business requirements are satisfied, the IT solution is directly derived through a process of model refinement. If models do not contain all the required technical details or contain errors, the derived implementation would be incorrect and the BDD lifecycle would have to be repeated. In this project we present a functional domain specific language embedded in Haskell, with which: 1) models can rapidly be produced in a concise and abstract manner, 2) enables focus on the specifications rather than the implementation, 3) ensures that all the required details, to generate the executable code, are specified, 4) models can be transformed, analysed and interpreted in various ways, 5) quality assures models by carrying out three types of checks; by Haskell's type checker, at construction-time and by functions that analyse the soundness of models, 6) enables users to define quality assured composite model transformations.

1. Introduction

Business process models are produced by business analysts to graphically communicate the business requirements to IT specialists. As business processes are updated to meet the new demands in the competitive market, the underlying IT solution is adapted, to reflect precisely the current goals of the organisation. The models should then act as an abstract representation of the solution. It is essential to adapt to Business-Driven Development (BDD) [14, 11] whereby models are refined into the IT solution and implemented in a Service-Oriented Architecture. This means that models must be free from data and control-flow errors, such as deadlocks (whereby a process waits indefinitely for some data or operation to complete). If models are not quality assured at the modelling phase, errors would be discovered later and the entire BDD lifecycle would have to be repeated. Combining model transformations with quality assurance would help modellers to preserve the correctness of models and rapidly carry out modifications [10].

Although various modelling languages have been developed to assist modellers in the production of high quality business process models, none of them adopted a functional approach, based on higher-order logic. As BDD is being adopted by most organisations, the need for such a language is becoming more evident. Since specialized functionality is required, a general-purpose language is not really necessary. Instead, a domain specific language, which provides the right abstraction and captures precisely the semantics of the business process modelling domain, must be developed. The definitions of the models would be easy to comprehend and reason about, by anyone who is not necessarily an IT specialist. However, since programming languages are made up of domain independent and dependent linguistic components, it is more cost effective and feasible to embed the new language in a general-purpose one. By defining its terms and type system as a domain specific library, the tools and features of the chosen host language, would be inherited by the embedded language [3, 4]. In this way, the language designers are able to reuse the infrastructure of the host and thus focus more on the semantics of the new language. Since the limitations of the host are also inherited, then it is important for the language designer to choose the appropriate host to embed the required language for that specific domain. Over the past years, Haskell [6] has been chosen as the host to embed languages for domains such as financial contracts [7] and hardware description [1]. As illustrated in [5], Haskell results to be an appropriate language to provide the right modularity and abstraction to develop a language which is maintainable, extendible, easy to design and easy to use even by non-programmers.

In this project we present a domain specific language embedded in Haskell, to model, transform and quality assure business processes in BDD. By adopting a functional approach, we developed a language: 1) with which various models can *rapidly* be produced in a *concise* and *abstract* manner, 2) allows users to focus on the required *behaviour* rather than the implementation, 3) ensures that all the required details, to generate the *executable* code, are specified, 4) the abstract representation can be *transformed, analysed and*

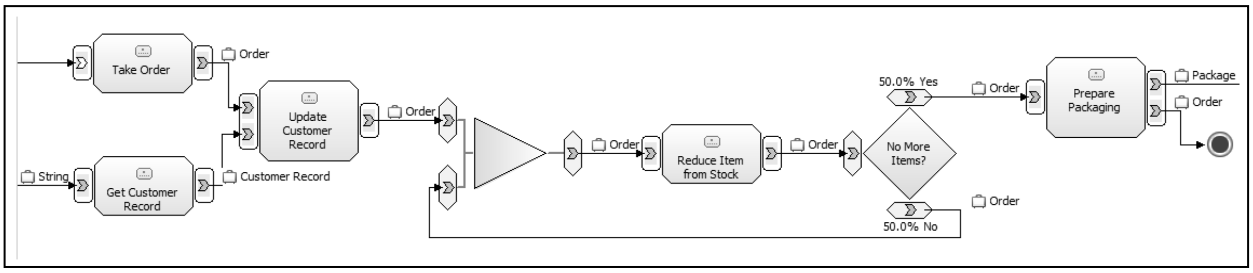

Figure 1. A process to handle orders, constructed using IBM WebSphere Business Modeler Advanced v6.0.2

Once the order is taken and the customer record is retrieved, the record is updated and the ordered items are reduced from the stock. The items are packaged and returned to the customer. The order is then discarded.

interpreted in various ways, 5) *quality assures* models by carrying out three types of checks; by *Haskell's* type checker, at *construction-time* through our embedded type system, and by specialised functions that *analyse* the soundness of models, 6) enables users to define new quality assured *composite model transformations*. With this language, we aim to capture the domain semantics of IBM's WebSphere Business Modeler Advanced v6.0.2[1] (WSBM).

2. Business Process Modelling

In process modelling, a sequence of business activities, with clearly defined inputs and outputs, is specified in a particular order, with the aim of capturing the business' requirements and objectives. Such models can represent the current (*'as is'*) and the future (*'to be'*) processes of the organisation. By analyzing these models, the efficiency and the quality of the processes can be improved before they are implemented.

As shown in Figure 1, in IBM's modelling tool, tasks (activities) are represented as boxes, a decision as a diamond shape, a merge as a triangle and a stop node as a black circle. User-defined business items (e.g. *'Order', 'Customer Record' or 'Package'*), basic typed items (e.g. *String* to represent the customer identification code) or control (e.g. the input to task *'Take Order'*) can flow along the connectors between the elements.

3. Embedding Business Process Models

Our language is essentially a library of Haskell modules, which provide the basic elements to construct any model and carry out operations on them. Since, based on some input, processes and modelling elements carry out some specific behaviour and produce some output, we kept with the style of the host and defined them as functions.

Before defining a model such as Figure 1, the business items specific to that business domain must be specified. Once done, it is then possible to define the tasks, as shown below:

```
tGetCustRec = task ”Get Customer Record”
                        (bvTString :-> biTCustRec)
```

To indicate the input and output types of the task, first class objects representing types are used. In this case, `bvTString` refers to a basic value of type *String* (as an input) and `biTCustRec` refers to a user-defined business item of type *Customer Record* (as an output). To distinguish between the types of our language and that of the host, the names assigned to all of our types, include a '`T`'. Thus, the type *String* in our language is referred to as `TString` rather than `String`.

Once all the tasks are defined, it is then possible to define the model as illustrated in Listing 1[2]. Note that `eNoMoreItems` and `eMoreItems` are boolean expressions which given an *Order*, decide how the flow should be diverted on the outgoing branches of the decision *'No More Items?'*.

The properties of the decision branches are defined using `branchProp` and include the expression defining when the branch is true and the probability that that branch would be true.

3.1. Strongly-Typed Process Fragments

Since the modelling elements in our language are essentially functions with specifically typed inputs and outputs, we can use Haskell's type checker to check the type-safety of the models at construction-time. Thus, if an element that expects as input some data item other than a *Customer Record*, is attached to the output of task *'Get Customer Record'*, the type checker would generate an error at construction-time and prohibits the user from carrying out other operations on that model.

This is possible through the use of phantom types [16] in the definitions of the provided basic modelling elements (such as `task`, `exclDecision`, `merge`, `stop`). Since the defined models need to be interpreted and

[1]http://www.ibm.com/developerworks/websphere/zones/businessintegration/roadmaps/modeler/roadmap_advanced.html

[2] In the definition, note that prefix '`t`' is used for a task, '`e`' for an expression and '`o`' for output

```
pfOrderHandling (x,y) =
                let  otUpdateCustRec = tUpdateCustRec (tTakeOrder x, tGetCustRec y)
                     omerge = merge (otUpdateCustRec, oMoreItems)
                     (oNoMoreItems, oMoreItems) = exclDecision "No More Items?",
                                                  (branchProp eNoMoreItems 0.5, branchProp eMoreItems 0.5)
                                                  (tReduceItemFromStock omerge)
                     (otPreparePackaging_Package, otPreparePackaging_Order)= tPreparePackaging oNoMoreItems
                      ostop = stop otPreparePackaging_Order
                in   (otPreparePackaging_Package, ostop)
```

Listing 1. Defining the process fragment in Figure 1 using basic modelling elements in our language

```
pfOrderHandling = (tTakeOrder -|- tGetCustRec) ->>- tUpdateCustRec ->>-
                    soundCycle tReduceItemFromStock ("No More Items?",
                               (branchProp eNoMoreItems 0.5, branchProp eMoreItems 0.5))
                  ->>- tPreparePackaging ->>- (id -|- stop)
```

Listing 2. Defining the process fragment in Figure 1 using connection patterns in our language

analysed in various ways, we have opted for a deep embedded approach, such that, once the model is defined and type checked, an internal abstract representation made up of primitive untyped constructors is defined. In this way, any model in our language can be interpreted and analysed using the same functions.

Type classes are also used extensively to carry out various computations and checks at the type level as discussed in [8].

3.2. Detecting Sharing and Loops

An issue encountered, while analysing structures in deep embedded languages, is the inability to detect shared fragments and loops. Shared fragments are usually those whose output is used as an input to more than one fragment. Loops are usually present in fragments such as Figure 1, where the output of an element is used as an input to another previous element. In such situations, fragments are evaluated more than once, or in case of loops, evaluated until it runs out of memory space. To be able to detect sharing and loops, we used non-updateable references as proposed in [2].

3.3. Packaging Models into Sub-Processes

By defining process fragments as functions, as shown in Listing 1, details of the model are abstracted away, such that it is easier for the user to reason about and define more complex models. The only problem is that, during analysis, such blocks are not identified. To mitigate this issue, fragments in our language can be packaged into a sub-process, such that during analysis, the interpreter can identify the block of elements and decide either to consider this sub-process as one single modelling element or explore its internal elements. Different from fragments defined as functions, the inputs and outputs of sub-processes are filters, such that only data is allowed. Thus, if Figure 1 is packaged, the sub-process would take a *String* as input and produce a *Package* as output. The control input, which is required for task *'Take Order'*, would be derived from the input data flow. To identify such blocks, the user must explicitly tag the sub-process.

3.4. Connection Patterns

Languages embedded in a host that supports higher order functions, usually provide connection patterns to ensure the production of concise, elegant, readable and easy to comprehend definitions. These patterns are essentially functions, which given other functions as input, combines them in a particular manner and returns a more complex one. As illustrated in Listing 2[3], such patterns in our language are also important to help users visualize the textually defined models. Different from Listing 1, the model in Figure 1 is 1) defined with essentially one line of code, 2) the inputs and outputs are never referenced and 3) it is easier for the reader to follow the order of execution of the elements in the model. For instance, `-|-` is used to compose fragments in parallel and `->>-` is used to serially compose elements and allow the system to infer underlying connection types. Different from the previous examples, `soundCycle` is a complex connection pattern, which abstracts the implementation of an entire fragment and ensures the production of sound cycles (that lack deadlocks and lack of synchronisation). After analysing different models and patterns identified in [9], libraries of such simple and complex connection patterns were defined in our language.

3.5. Parameterized Models

Different from the current modelling tools, in our language, users can define their own parameterized

[3] In the definition, the Haskell function `id` is used to allow the first output of the task *'Prepare Package'* to flow through without being modified

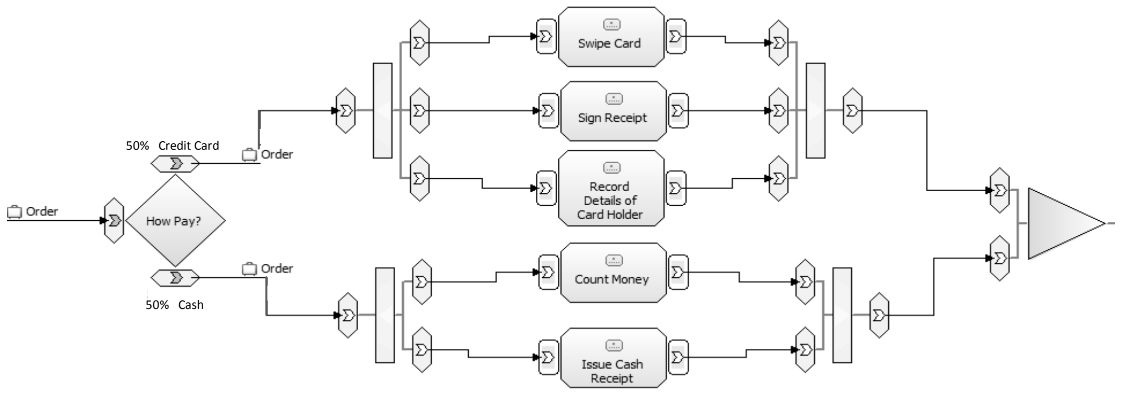

Figure 2. A decision-merge with internal fork–joins, constructed using IBM WebSphere Business Modeler Advanced v6.0.2

```
fork_joins [pfsFJ] = fork_join pfsFJ
fork_joins (pfsFJ : pfsFJs) = (fork_join pfsFJ) -|- (fork_joins pfsFJs)

decisionMerge_forkJoins nm brs pfsL = exclDecision_merge nm brs (fork_joins pfsL)
```

Listing 3. A parameterized model to define models such as Figure 2

`fork_joins` *generates the internal fork-join fragments by using the connection patterns* `fork_join` *and* `-|-` `decisionMerge_forkJoins` *constructs the actual fragment, where* `nm` *and* `brs` *are respectively the name and the properties of the branches of the decision.* `pfsL` *is the list of the process fragments for the internal fork-joins*

```
pf = decisionMerge_forkJoins  "How Pay?"
                              (branchProp eCreditCard 0.5, branchProp eCash 0.5)
                              [(tSwipeCard, tSignReceipt, tRecordDetailsCardHolder),
                               (tCountMoney, tIssueCardReceipt)]
```

Listing 4. Defining the model in Figure 2 using `decisionMerge_forkJoins` (Listing 3)

models, such that, families of similar structured process fragments can be composed. If a user identifies that a particular structure is repeatedly used, then it would be wise to define a parameterized model. In this way, by simply invoking one function and providing the appropriate input parameters, the required model would rapidly be constructed in an abstract manner. Thus, definitions using these models would be concise, readable and easier to comprehend.

Listing 3 is an example of a parameterized model which can be used to construct models such as Figure 2. Depending upon the input list of fragments, the required fork-joins are constructed and enclosed between a decision and a merge. In this way, the complex model in Figure 2, can rapidly and safely be constructed by the definition in Listing 4.

4. Model Transformations & Quality Assurance

Since our language is based on higher order logic, it is possible for users to declaratively define pre and post conditions and composite transformations. A number of basic checks and transformations are provided in our language as functions, such that similar to functional composition, these can easily be composed into more complex checks and transformations, as shown in Listing 5.

This complex transformation is made up of two simpler ones (`transf1`, `tranf2`), which are carried out in sequence. The first is a branching type transformation. It uses the provided basic checks, to define pre-conditions and to decide which transformation should be carried out. Thus, `transf1` does the following: if a sub-process named *"Order Verification"* is found, it is *renamed* to *"Certify Order"*; else, if the process contains a task/s named *"Reject Order",* the first one is *substituted* with another task named *"Apply Special Terms to Order"*. `transf2` then *renames* decision *"Is Order Valid?"* to *"Is Order Certified?"*. Thus, the basic checks `containsSubProcess` and `containsTask`, and the basic transformations `renameSubProcessQA`, `substituteTaskQA` and `renameDecisionQA` are used. As indicated by the suffix '`QA`', these basic transformations are quality assured. This means that other pre and post conditions are internal defined, such that, the basic transformation is not carried out and the transformed model is not returned unless the conditions are satisfied. An important condition is the assurance that a model is structurally correct and sound before and after the transformation is carried out. In this way, by combining model transformations with quality assurance, modellers can

```
tApplySpecialTerms = task "Apply Special Terms to Order" (biTOrder :-> biTOrder)

transOrderProcessing pf x =

    let (hasSPOrderVerif, _) = containsSubProcess "Order Verification" pf
        (hasTaskRejectOrder, _) = containsTask "Reject Order" pf

         transf1@(wasTransDone, transMsg, transPF)  =
              if (hasSPOrderVerif)
                   then (renameSubProcessQA "Order Verification" "Certify Order" pf x)
                   else if (hasTaskRejectOrder)
                         then (substituteTaskQA "Reject Order" tApplySpecialTerms [1] pf x)
                         else (Succeeded, "", pf x)

         transf2 = renameDecisionQA  "Is Order Valid?"  "Is Order Certified?" pf x

    in   transf2
```

Listing 5. Defining the quality assured composite transformation `transOrderProcessing`

preserve the correctness of models and rapidly carry out the required modifications.

If on the other hand, the language should be extended with other basic transformations or checks, an appropriate recursive function, that pattern matches and handles the internal constructs, should be defined. Other basic checks can also be defined by carrying out analysis on the generated directed graph for the model.

5. Evaluation and Case Studies

A number of models created with WSBM have been used as case studies to evaluate our language. These models were constructed using different approaches and each one was analyzed.

The first two case studies are based on two models obtained from the sample projects that are available with IBM's tool. These projects are very realistic and they were purposely created to help modellers learn how to use IBM's tool. Thus, it was thought that these models would be ideal to evaluate our language and help modellers learn how to define real world processes in our language. In fact, these samples projects are also provided as samples in our language. The main aim of the first case study was to analyse the different ways how models and modelling elements can be defined using our language, and which of these, would be most feasible, for a modeller who is not an IT specialist and who might already be familiar with IBM's modelling tool. The main aim of the second case study was to identify how easy a complex model can be defined, with the least amount of effort, components and expertise, while still ensuring the correctness of the model. Connection patterns played a very important role to provide the required abstraction and modularity to handle such complex models. The third case study considered a model which was intentionally constructed to illustrate the importance of connection patterns to handle some of the most commonly modelled fragments and other fragments, which can easily introduce new errors, if constructed manually. Finally, two examples of parameterized models were investigated in case study 4.

After evaluating these case studies, it was evident that, using our language, any business process model can rapidly be constructed in a concise and readable manner. This was possible through the use of connection patterns and parameterized models that allowed us to achieve the required modularity and abstraction. Moreover, the produced models were guaranteed to be of a high quality. Through our embedded type system, errors were identified as early as construction-time, when the script defining the model was compiled. In this way, errors were trapped at the modelling phase and were not allowed to propagate to the succeeding stages in BDD lifecycle.

These case studies enabled us to identify the effectiveness of this first prototype of our language. Other more comprehensible evaluation techniques, which would employ more domain experts and analyse a wide variety of models, shall be carried out in the next version.

6. Related Work

To assist modellers, various languages and tools, such as WSBM, having been developed. The most recent is Business Process Modelling Notation (BPMN) [15], whose main objective is to unify the features of all the other languages. Still, none of the languages adopt a functional approach, based on higher-order logic.

As argued in [10], a *declarative approach* would be appropriate to define composite transformations and pre and post conditions that assure the quality of the produced models. In [12], pre and post conditions of out-place transformations were represented in the Object Constraint Language and used successfully to refine the models into the executable BPEL code. However, such an approach brings about other advantages. Noting how effectively certain features in Haskell [6] were used to define circuits [17] and other domains, we were inspired to use Haskell as our host, and thus define models as functions.

To analyse and interpret the model in an infinite variety of ways, we have adopted a *combinatorial approach*, as in [7] whereby a combinator library in Haskell was produced to compose financial contracts. By employing such a deep embedded approach, the basic modelling elements in our language act as combinators.

To extend the WSBM in [10], IBM presents a *model transformation framework*. Their main objective is to provide an abstract layer over the tool, such that specialized developers would be able to easily define new transformations, quality assure them and integrate them into the tool. However, since it uses first-order logic, developers still need to consider the implementation of the required operations. Moreover, to carry out checks while the user is constructing or editing the model, linear-time algorithms that do not introduce any significant delay, such as [18] would have to be adopted. In contrast, with our language, we are able to statically trap errors and ill-typed processes at construction-time through our embedded type system and Haskell's type checker. These are identified before any further computation is carried out. Phantom types and type classes are used in a similar way as in [13] and [1] to define our strongly typed system. Besides this, specialized functions, that operate on the abstract representation, are provided to analyze the structural correctness of the models.

Over the years, various *quality assurance techniques* have been suggested. In [18], the authors argue that if models are decomposed into Single-Entry-Single-Exit fragments, they can be quality assured more effectively by using linear-time control-flow heuristics or complete state analysis. Similarly, a set of patterns and anti-patterns have been identified in [9]. To help modellers rapidly and safely transform the current *'as-is'* to the future *'to-be'* models, in-place model transformations must be combined with quality assurance techniques. Even though IBM's framework enables programmers to define such transformations, it is still based on first-order logic and thus, it not possible for the modellers themselves to create composite branching and iterative transformations and to define pre and post conditions that quality assures them. In our language, users can declaratively define sequential, branching and iterative composite transformations and the required pre and post conditions.

7. Conclusion

With our functional modelling language, we have managed to develop a language which is able to capture precisely the domain of business process modelling and allows users to model, transform and quality assure business processes in BDD. Connection patterns play an important role to ensure that the definitions of models are readable, easy to comprehend and type-safe. Different from other previous modelling tools, users are able to define their own parameterized models and transformations. By defining and using the provided quality assurance checks, the soundness of the processes is guaranteed and thus the derived IT solutions should be correct. Quality assurance can be combined to model transformations and by using the generated directed graph for the model, users can easily analyse the processes. Since our language has been successfully embedded in Haskell, we were able to adopt a functional approach and inherit the infrastructure, tools and features of the language without necessarily having to re-implement them. Various models have been defined in our language to ensure that our objectives were achieved. In the next version, we would like to include parameterized verification and pass on the defined processes to some model checkers to carry out complete state analysis.

References

[1] K. Claessen, *Embedded languages for describing and verifying hardware*, Dept. of Computer Science and Engineering, Chalmers University of Technology, Ph.D. thesis, April 2001.

[2] K. Claessen and D. Sands, "Observable Sharing for Functional Circuit Description," *Proceedings of Asian Computer Science Conference (ASIAN)*, Springer Verlag, 1999, p. 12.

[3] P. Hudak, "Building domain-specific embedded languages," *ACM Computing Surveys*, 1996, p. 196.

[4] P. Hudak, "Modular domain-specific implementation and exploration framework for embedded software platforms," *DAC '05: Proceedings of the 42nd annual conference on Design automation*, 1998, pp. 254-259.

[5] P. Hudak and M. P. Jones, "Haskell vs. Ada vs. C++ vs. Awk vs. … - An Experiment in Software Prototyping Productivity," Yale, 1994.

[6] S. P. Jones, *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press, 2003.

[7] S. P. Jones, J.-M. Eber, and J. Seward, "Composing contracts: an adventure in financial engineering," *ACM SIG - PLAN Notices*, 2000, pp. 280-292.

[8] S. P. Jones, M. Jones, and E. Meijer, "Type classes: exploring the design space," *Proceedings of the Haskell Workshop 1997*, 1997.

[9] J. Koehler and J. Vanhatalo, "Process anti-patterns: How to avoid the common traps of business process modeling, Part 1 modeling control flow, Part2 modeling data flow," *IBM WebSphere Developer Technical Journal 10.2, 10.4*, 2007.

[10] J. Koehler, et al., "Combining Quality Assurance and Model Transformations in Business-Driven Development," *Proceedings of Applications of Graph Transformations with Industrial Relevance 2007*, 2007, pp. 1-16.

[11] J. Koehler, et al., "The role of visual modeling and model transformations in business-driven development," *Proceedings of the 5th International Workshop on Graph Transformation and Visual Modeling Techniques*, Elsevier, 2006, pp. 1-12.

[12] J. Koehler, R. Hauser, S. Sendall, and M. Wahler, "Declarative techniques for model-driven," *IBM Systems Journal, vol. 44, no. 1*, 2005, pp. 47-65.

[13] D. Leijen and E. Meijer, "Domain specific embedded compilers," *Proceedings of Domain-Specific Languages*, 1999, pp. 109-122.

[14] T. Mitra, "Business-Driven Development", *IBM developerWorks article*, 2005.

[15] OMG, *Business Process Modeling Notation Specification 2008, Version 1.1*, Object Managementt Group (OMG), 2008.

[16] M. Rhiger, "A foundation for embedded languages," *ACM Trans. Program. Lang. Syst.*, 2003, pp. 291-315.

[17] M. Sheeran, "Hardware Design and Functional Programming: a Perfect Match," *j-jucs*, vol. 11, no. 7, 2005, pp. 1135-1158.

[18] J. Vanhatalo, H. Völzer, and F. Leymann, "Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition," *Service-Oriented Computing – ICSOC 2007*, 2007, pp. 43-55.